CITI Technical Report 00-8

An analysis of the TUX web server

Chuck Lever, Sun-Netscape Alliance
chuckl@netscape.com

Marius Aamodt Eriksen, Linux.com
marius@linux.com

Stephen P. Molloy, University of Michigan
smolloy@engin.umich.edu

ABSTRACT

We report on a high-performance in-kernel web server for Linux known as the Threaded linUX http layer, or TUX, for short. TUX uses aggressive network layer data caching to accelerate static content delivery, and invokes CGI scripts directly from the kernel to accelerate dynamic content generation. We describe the TUX web server architecture, modifications included in the patch, and how they affect kernel operation and web server performance.

November 16, 2000

Center for Information Technology Integration
University of Michigan
519 West William Street
Ann Arbor, MI 48103-4943

This document was written as part of the Linux Scalability Project. The work described in this paper was supported via grants from the Sun-Netscape Alliance, Intel, Dell, and IBM. For more information, see our home page.
If you have comments or suggestions, email <linux-scalability@citi.umich.edu>

An analysis of the TUX web server

Chuck Lever, Sun-Netscape Alliance
chuckl@netscape.com

Marius Aamodt Eriksen, Linux.com
marius@linux.com

Stephen P. Molloy, University of Michigan
smolloy@engin.umich.edu

1. Introduction

As the demand for faster and more scalable web service increases, system designers have discovered ways to improve web server performance and scalability by integrating web server functionality into operating systems. This trend began when O/S designers added system interfaces specifically designed for web servers, such as the *TransmitFile* system call in Windows NT [9].

Ideally, one could create an O/S whose only purpose is to provide HTTP access to a local file system and run CGI scripts securely on behalf of web clients. Data is cached in the kernel's address space with zero-copy techniques to reduce the overhead associated with data copying operations and checksum computation. CGI scripts could be invoked directly by the kernel, and output from the scripts routed directly to the network layer via zero-copy to reduce data copying and context switching.

TUX is an HTTP protocol layer and a web server object cache integrated into the Linux kernel. Its aim appears to be to approach the ideal of a fully integrated web server and operating system [2]. TUX stands for Threaded linUX http layer. Created by Ingo Molnar, an employee of Red Hat Software and long-time Linux kernel developer, TUX takes the next step in web server evolution that TCP took nearly a decade ago when it was integrated into UNIX kernels as a standard feature of networking stacks [4].

The TUX web server competed favorably in a web server competition earlier this year, performing better than web servers of more standard design including Microsoft's IIS on Windows NT [3]. TUX uses several general techniques to achieve high performance, such as:

- SMP-friendly multi-threading to complete complex requests asynchronously
- Driving the web server directly from the kernel's networking layer to create a truly network event-driven server
- Caching complete responses in the kernel's networking layer to accelerate static content delivery
- Providing a secure interface for generating simple dynamic content quickly from within the kernel
- Providing a rich interface for user-level web applications to generate complex dynamic content in a user-level context

Molnar predicts it will be easy to adapt TUX's high performance architecture to other operating systems, network protocols, and applications [4].

In this paper we describe the TUX web server architecture, modifications included in the TUX Linux kernel patch, and how they affect kernel operation and web server performance. We cover its basic features in Section 2. Following sections discuss specific areas of its design, including its event and threading models, and its architectural motivations. Those interested in performance comparisons should consult the SPEC report [3].

2. Basic Features

TUX serves files directly out of locally accessible file systems, including files in ext2, NFS, and DOS FAT file systems mounted under the server's document root. It provides a simple and secure programming interface, called HTTPAPI, that supports generating content dynamically in kernel modules. The interface allows C language kernel modules easily to access files and other resources from within the kernel environment.

TUX triggers *external CGIs* via HTTPAPI. An external CGI is a script that generates dynamic content from outside the kernel, usually started via a fork/exec pair in traditional web servers. The HTTPAPI dynamic interface provides enough functionality to exec scripts with `stdin` and `stdout` set up as sockets connected to the client. The logic to start external CGIs is an example of how to use HTTPAPI, and is included with the TUX patch. See net/http/extcgi.c for details.

CGI scripts may also be triggered by user-level activity. This is referred to as a *user-level CGI*. The new `http()` system call (see below) allows user processes to interact with the kernel web server.

2.1 Protocol support

TUX supports the basic functions of HTTP version 1.0 and 1.1. It directly supports the `Cookie:` and `Connection:` fields. Everything else that is optional in the HTTP specification is ignored [10]. Specifically, it ignores optional headers such as `Last-Modified-Time:` so it does not support efficient client-side and proxy caching yet.

TUX does not need another in-kernel server such as `khttpd`, nor does it require a backing server such as Apache [11, 5]. It handles server-side includes correctly to support SPECweb99 dynamic content benchmarking. Optionally, however, a backing server can handle any request TUX doesn't recognize, such as when it can't parse HTTP header information in an incoming request. Headers it doesn't recognize as required or optional will cause the parser to pass the request to a backing server via a fast socket redirection mechanism. No modifications to the backing server are required to handle the redirected requests. The backing server must run on the same system as TUX.

2.2 Configuring TUX

Configuration starts with the kernel build process. After applying the patch, new options appear in the kernel's build configuration menu program allowing selection of several TUX build-time options. The options appear in the "Networking options" submenu when "TCP/IP Networking" is enabled. TUX build options include:

- Threaded linUX HTTP layer (TUX) – this enables the following options, and causes the kernel build process to compile in the TUX code base. Specifying a module here rather than building it in disables the following options. Addition of this feature is controlled by the `CONFIG_HTTP` macro.
- CAD and CAD2 modules – this enables the inclusion of support for the SPECweb99 CAD dynamic application via trusted HTTPAPI modules (HTTPAPI is described in further detail later in this report). Addition of this feature is controlled by the `CONFIG_HTTP_CAD` macro.
- External CGI module – this enables support for starting CGI programs from within the kernel. It can be either built in to the kernel or be built as a separate kernel module. Addition of this feature is controlled by the `CONFIG_HTTP_EXTCGI` macro.
- debug TUX – this enables debugging traps and extra kernel log output for monitoring TUX during operation. This featured is enabled by defining the `CONFIG_HTTP_DEBUG` macro.

Dynamic configuration of the server is available via Linux's `sysctl` mechanism. This provides a group of files in the /proc file system that, when read, report the values of kernel variables, and when written into, modify variables in the kernel address space.

Configurable parameters include:

- *Document root* – where the root directory for exported web documents resides. The default value is /var/www/http.
- *Log file* – where the server's log file resides. The default value is /var/log/http.
- *Starting and stopping web service* – writing a one into /proc/sys/net/http/stop causes web service to stop. Restarting and unloading are currently unsupported.
- *Redirect port* – which port the backing server is listening to. The default value is port 8080.
- *Logging* – whether the web server is writing log output to its log. The default is not to write log output.
- *Kernel logging* – whether the web server is writing verbose debugging output into the kernel's log. The default is not to write log output.
- *Reporting thread count* – configures how many web server threads will start next time web service is started.
- *Listening port* – which port the TUX web server is listening to. The default value is port 80.

- *Maximum connections* – how many connections the web server will maintain with clients at once. The default value is 10,000 connections, but this parameter is currently ignored.
- *Maximum backlog* – how many connections can wait in the server's listen backlog queue before new connection requests are refused. The default value is 2048 connections.
- *Keepalive timeout* – how many seconds to keep connections with no activity alive. The default value is not to maintain a connection keepalive timer.
- *Maximum cached file size* – how large a file can be maintained in the web server's cache. The default value is 100,000 bytes. This prevents a single streaming media file from consuming the entire web object cache.
- *Mode of forbidden files* – configures a file mode mask that forbids files to be accessed. The default is not to deny any file modes.
- *Mode of allowed files* – configures a file mode mask that allows files to be accessed. The default is to allow access to files that are readable by "others."
- *Mode of user-space modules* – configures a file mode mask that allows user-space modules to be executed. The default is to allow execution of user-space modules that are executable by "user." The setuid and setgid bits are also required for user-space modules.
- *Mode of external CGI modules* – configures a file mode mask that allows external CGI modules to be executed. The default is to allow external CGI modules to be executed if any execute bit is set. Execution of external CGI modules requires the kernel to be built with that option enabled (see above).
- *Enable input and output packet delay timers* – how long to delay packets in the network layer. The default value is not to delay packets.
- *Disable extra `sk_buff` copy operations* – whether to use `sk_buff` copy operations for multi-fragment network buffers. The default is to use the old mechanism, which copies the buffer.

Server logs are generated in binary format. TUX's designers feel that writing logs in a binary format saves I/O bandwidth and disk space over writing human-readable logs in W3C ASCII format. Each log record contains all information that can be referenced from an `http_req_t` struct.

Worker threads share a common ring buffer for writing log entries. The entries are padded to cache line boundaries, ensuring no false cache line sharing between threads on separate CPUs. A separate thread flushes log buffers every second or when 95% full. If the buffer fills entirely, worker threads block until existing log data is written to disk. These limits are compile-time constants, and cannot be changed during server operation.

2.3 User-level access to the kernel's HTTP layer

User processes drive TUX servers. The TUX patch adds a new system call, `http()`, which provides an interface between user-level activities and kernel-provided network functions. User processes identify themselves as web service threads, and then use the `http()` system call as an up-call mechanism to accept work from the kernel. These threads also act as the main listener threads for scheduling work within TUX.

At this time there doesn't appear to be any access control logic built into this system call.

The caller specifies one of the following actions:

Startup or shutdown – start or stop web services exported via the TUX server in the kernel.

Register or unregister modules – add and remove kernel modules from the list of available in-kernel dynamic content generation modules.

Start or stop listener threads – start another listener thread or stop an existing listener thread, including self.

Set date in outgoing headers – set the text-format date that appears in the `Date:` header in outgoing responses.

Pass through the thread's housekeeping event loop – check work queues, then sleep until something interesting happens. This allows `http()` to function as an up-call to pass work from the kernel up to the user-level.

Finish a request – write a log entry if necessary and make ready for the next event.

Prime the cache – be sure the requested object is in the cache. This schedules a read operation, if necessary.

Read an object's data – read data from an object represented by a URL into a user-level address space.

Send an object's data to a client – transmit data from an object represented by a URL to a client.

```
typedef struct user_req_s {
        int http_version;
        int http_method;
        int sock;
        int bytes_sent;
        int http_status;
        unsigned int client_host;
        unsigned int objectlen;
        char query[MAX_URI_LEN];
        char *object_addr;
        char objectname[MAX_URI_LEN];
        int module_index;
        char
modulename[MAX_MODULENAME_LEN];
        char post_data[MAX_POST_DATA];
        char new_date[DATE_LEN];

        int cookies_len;
        char cookies[MAX_COOKIE_LEN];

        int event;
        int thread_nr;
        void *id;
        void *private;
} user_req_t;
```

Figure 1. User API request object.

This structure is exposed to user-level applications, and is one argument of the `http()` system call. The structure passes information into the system call, and also acts as a buffer for return values.

2.4 Availability and code readiness

TUX is available as a patch to Linux kernel version 2.4.0-test8, however the patch doesn't apply directly to this kernel version. The patch applies to a variant of a pre-release version of 2.4.0-test8 that Molnar was working on when he generated this patch. We applied the patch to 2.4.0-test8 and modified the net/core/dev.c file by hand to compile the kernel.

The TUX patch contains optimizations to some network device drivers that increases buffer size, fine-tunes PCI bus management and error-checking, aligns important data structures to first-level CPU cache line boundaries, and provides instruction-level locking for better SMP behavior. These changes are not specific to TUX's functionality.

Support for Intel 32-bit processors is obvious in the patch. It is not clear whether TUX has been tested or runs on other Linux-supported hardware platforms.

TUX is not feature-complete at the time of this writing. Support is incomplete for virtual hosting, response headers that provide information for efficient proxy caching, and byte range transfers. In addition, because it applies only to development versions of the Linux kernel, we surmise that it has not reached production-readiness.

TUX is available from Red Hat as part of their Enterprise 7.0 distribution. The patch, with a few minor changes, has been incorporated into the kernel provided with 7.0, and now includes documentation in SGML format, and all the user-level pieces. It is Red Hat's usual policy to include any patches they make to the kernel source with the distribution.

You can find information on the TUX web server, including documentation and the source code for the user-level pieces here:

`http://www.redhat.com/products/software/linux/tux/`

3. Accelerating Static Content

Copying data and calculating checksums on outgoing network packets are possibly the two most expensive operations in serving static web content already in memory. A typical method to reduce these expenses is to cache outgoing responses on the server in the hope that another client will request the same object again soon.

To keep cached checksum values valid, data must be cached in kernel network buffers (known in Linux as `sk_buffs`) rather than as file system data or in some application level cache. TUX accomplishes this by adding references to cached data to the kernel data structures that represent in-core inodes.

3.1 Managing network buffer fragmentation and reuse

The Linux kernel uses `sk_buff` structures to manage data that is going to and from network devices. Metz discusses the differences between BSD-style `mbufs` and Linux `sk_buffs` [7].

The TUX patch improves the ability of the Linux networking layer to manage dynamically changing `sk_buffs`. Hagino documents several `mbuf` fragmentation issues he found while implementing IPv6 and IPsec on 4.4BSD [6]. Generally these buffers become fragmented due to sophisticated protocol processing, which requires unpredictable changes to a buffer's length while it is being prepared for transport.

Molnar introduces a new data structure, called `skb_frag_t`, to manage buffer fragmentation and allow scatter/gather I/O using `sk_buffs`.

```
struct skb_frag_struct {
        unsigned int csum;
        int size;
        struct page *page;
        int page_offset;

        void (*frag_done)
               (struct sk_buff *skb,
               skb_frag_t *frag);
        void *data;
        void *private;
};
```

Figure 2. Data type for managing buffer fragments.

Every `sk_buff` now points to up to four buffer fragments. Each fragment can be as large as the system page size (4096 bytes on Intel processors). The checksum for each fragment is maintained independently, so the server can reuse fragments without recomputing the checksum of each fragment.

This data structure refers to cached checksum data:

```
struct csumcache_struct {
        int size;
        int packet_size;
        skb_frag_t *array;
};
```

Figure 3. Simplified checksum cache object

The `size` field contains the number of packets in the fragment array pointed to by the `array` field. A `csumcache_struct` can point to an arbitrary number of buffer fragments, unlike a normal `sk_buff`, thus allowing efficient manipulation of the length and contents of arbitrarily complex network buffers, similar to aggregates in IO-Lite [12].

This also allows portions of a response to be cached and checksummed independently. When constructing a response to a client request, the server need only pull together previously cached portions and add their checksums. Scatter/gather I/O obviates an additional copy operation.

TUX maintains response headers and server-side include data in separate buffer fragments. Processing a server-side include splits a single fragment into three. Otherwise file data fragments are up to a page (4096 bytes on ia32) in length. This means buffer fragments can pull file data directly out of the page cache without an extra copy operation.

3.2 Linking URLs to cached data

A URL object is created in `lookup_urlo()` to bind an in-core inode to data cached in the checksum cache. There is no separate URL object cache; TUX looks up URL objects by searching the directory entry cache for appropriately named file objects.

The directory entry cache is managed via a hash table whose size is determined by the physical memory size of the hardware where Linux is running. Generally entries in the cache are not reclaimed – the cache is allowed to grow while there is still free memory. As memory becomes constrained, the system page allocator invokes a cache pruning operation that removes least recently used entries from the directory entry cache. Pruning a directory entry can also remove an in-core inode and any related data in the page cache.

```
struct urlobject_struct {
        csumcache_t *csumc;
        struct inode *inode;
        atomic_t users;
        struct list_head
               secondary_pending;
        int header_len;
        int body_len;
        int filelen;
        int SSI;
        tcapi_template_t *tcapi;
        atomic_t csumcs_created;
        struct address_space_operations
               *real_aops;
};
```

Figure 4. Kernel-level cache object.

The `tcapi` field identifies a dynamic trusted module by defining its entry points and properties. HTTPAPI templates are described later in this report.

Linux uses an `address_space` structure to denote the set of mappings between an inode's data pages and one or more process address spaces. This structure also contains a vector of virtual functions for moving an inode's data into and out of the system's page cache. These functions include `readpage`, `writepage`, `sync_page`, `prepare_write`, `commit_write`, and `bmap`.

Address-space operations are replaced for inodes that represent cached web server objects. The old operations vector is maintained in the URL object in the `real_aops` field. The old operations vector is restored when TUX removes its last reference to an inode.

The new virtual functions act as a wrapper around the original address space map operations. For the most part, the wrapper functions perform only error checking and argument marshalling.

The `readpage` wrapper is more sophisticated, however. It directly links the checksum cache to the inode's data in the page cache. Whenever a read request occurs, the relevant data pages are mapped into the kernel's address space and copied into the checksum cache data structure (see Figure 3) associated with the inode via its URL object.

In summary, TUX provides a new set of data structures that allow it to cache partial and complete responses, with checksum, in the kernel's network layer. Cached web objects are managed as a part of the kernel's LRU directory entry cache.

4. Accelerating Dynamic Content

TUX accelerates dynamic content generation by providing two new interfaces. The user-level interface, the new `http()` system call, is described earlier. This section describes the kernel-level interface.

4.1 The HTTP dynamic API

TUX provides an interface, called HTTPAPI that supports the generation of dynamic content by trusted kernel modules. This interface is described in the file net/http/HTTPAPI.txt, and is summarized here. This dynamic API is intended for simple, oft-used requests with few security issues. Complex and slow requests should be handled in user-space.

These modules are invoked using a special URL of the form:

```
http://server.your.domain/module?argument
```

where `module` is the unique name of the dynamic module to be invoked, and `argument` is a text argument passed by the server to the module.

Every HTTP trusted dynamic module defines a `tcapi` template, which defines entry points and module properties. The template looks like this:

```
struct tcapi_template_s {
       char *vfs_name;
       char *version;
       int (*query) (http_req_t *req);
       int (*send_reply)
              (http_req_t *req);
       int (*log) (http_req_t *req,
              char *log_buffer);
       void (*finish) (http_req_t *req);
};
```

Figure 5. Dynamic module API template.

The `vfs_name` field refers to the unique name of the module. The `version` field refers to a string that names the version of the interface supported by the module, usually "`TUX 1.0`".

The other four fields are virtual functions implemented in each module. The server invokes the `query` function to handle a new request from a client. Modules can return a filename in the `http_req_t` struct that is then transmitted to the requesting client by the server. Otherwise, if the `send_reply` function pointer is not NULL, the server invokes it to generate a response. Module-specific log messages can be generated by the `log` function. When the server closes a connection, it invokes the `finish` function.

The interface makes available several other structures for use by module programmers. These are opaque descriptors that refer to files, URIs, pages, file system directory entries, and the server's document root.

A limited system-call-like API is also provided to simplify module programming. This allows modules to safely open, close, look up, read, write, and mmap files under the server's document root, send buffers or files directly to clients, retrieve file sizes and modification times, allocate and free "heap" memory, manipulate mutexes, exec new threads, sleep without blocking the kernel, or retrieve the client's IP address.

When cloning a thread to run a CGI, the kernel sets the thread's privileges to be nothing.

A separate interface, called `http_miss_req`, is available to trusted modules that wish to initiate an I/O operation to fill a cache slot. This can pre-fetch an object, or can schedule an asynchronous read if a cache look-up operation fails. When the I/O operation completes, an I/O thread queues the request onto one of the requesting thread's output queues, and wakes the thread.

Examples of trusted dynamic modules are included in the patch. These are the CAD and CAD2 modules that implement the CAD dynamic application required in the SPECweb99 benchmark. These applications are normally implemented in Perl, but SPECweb99 rules allow these applications to be rewritten. It is likely that simply rewriting such a script in C has an enormous impact on benchmark performance.

If a trusted module causes a program interrupt, the kernel stops the interrupting thread and prints a diagnostic on the system console. This is referred to as an "oops." Generally a system can continue functioning after this occurs, but sometimes a thread may have acquired a lock or other system resource that will cause the system eventually to crash or otherwise melt down.

In summary, HTTPAPI is a kernel module programming interface that supports the generation of dynamic content by trusted kernel modules. It provides a miniature system call-like interface that can fork, exec, manage file data, populate the web object cache asynchronously, or manage network connections. A kernel module that uses the HTTPAPI interface invokes external CGI scripts.

5. Threading Model

TUX uses a scheme similar to the FLASH web server to handle cached requests quickly via an event-driven mechanism, while managing any necessary disk I/O asynchronously in separate threads [8].

5.1 I/O threads

There are two thread pools. The first pool of threads is referred to as the *IO*, or *cache-miss* thread pool. These threads populate the cache asynchronously at the behest of listener threads. The number of cache-miss threads is a compile-time constant, currently 10 (a better number might be a multiple of the number of CPUs configured in the system). All cache-miss threads are created when the web server is initialized (system restart). There is no provision for starting or stopping these threads during normal operation, although they do respect signals.

Cache fill requests are queued onto a single global list via HTTPAPI's `http_miss_req()` interface. All cache-miss threads try to pull work off this list until it is empty, at which point they sleep. Queuing items onto this list causes a single sleeping cache-miss thread to be awakened (wake-one semantics avoid a thundering herd).

TUX is supposed to balance load dynamically among IO threads, based on the number of requests they have pending. We haven't found logic to do this, possibly NYI.

There are two types of cache miss: a primary cache miss is one that TUX can fill by reading the requested object directly into its web object cache; a secondary cache miss is one where TUX has identified the request as one that it can't fulfill itself, so it passes the request to its backing server via socket redirection.

5.2 Fast threads

The other thread pool is referred to as *fast*, or *listener* threads. These threads handle responses that are already cached. There can be at most 16 such threads.

Fast threads are created when an otherwise normal user-level process invokes the `http()` system call and identifies itself as a TUX thread. A per-thread context area is anchored in the process's task structure. The `thread` field points back to the task struct where this object is anchored. The `threadinfo` objects are statically allocated in a contiguous array. The `cpu` field contains an integer that represents the number of the preferred CPU for this thread.

The `userspace_req` field is filled in with a user-space request that will be processed by this thread.

Becoming a fast thread relies on the `used` semaphore to prevent race conditions. A started thread (that is, one that has become TUX thread) contains a 1 in the `started` field. When stopping, a thread waits for connections to finish by adding itself to the `wait_stop` waitqueue via the `stop` field.

```
struct http_threadinfo
{
       http_req_t *userspace_req;
       int started;
       struct semaphore used;
       struct task_struct *thread;
       wait_queue_t wait_event
              [CONFIG_HTTP_NUMSOCKETS];
       wait_queue_t stop;
       int pid;

       int nr_requests;
       struct list_head all_requests;

       int nr_free_requests;
       spinlock_t free_requests_lock;
       struct list_head free_requests;

       spinlock_t input_lock;
       struct list_head input_pending;

       spinlock_t userspace_lock;
       struct list_head
              userspace_pending;

       spinlock_t output_lock;
       struct list_head output_pending;

       spinlock_t redirect_lock;
       struct list_head
              redirect_pending;

       struct list_head finish_pending;

       struct socket *listen
              [CONFIG_HTTP_NUMSOCKETS];
       int listen_cloned
              [CONFIG_HTTP_NUMSOCKETS];

       char * output_buffer;
       int cpu;
       unsigned int __padding[16];
};
```

Figure 6. TUX per-thread context structure.

Each thread can listen on up to four sockets at a time. Listener socket structures are anchored in the `listen` field. Multiple threads listening on the same socket set the appropriate entry in `listen_cloned` to prevent multiple closes of the same socket upon shutdown.

The `output_buffer` field anchors a 256-page buffer that is only used by the CAD module in the current version of TUX. The buffer will likely be used for sendfile-style requests.

The lists and locks manage request flow in TUX. A detailed discussion follows in the next section.

There are a relatively large number of locks per thread. This is an attempt at reducing lock contention on SMP hardware. It's not clear whether the design started with a small number of locks, and observed contention prompted an increase, or whether the design included so many locks at the outset.

In summary, user-level threads call into the kernel to pick up new work. If they can't respond to a request immediately with a cached response, they queue the request to be handled by the I/O threads.

6. Event Model

In this section we describe how TUX achieves a true network-event driven dispatching model.

Linux sockets each have their own waitqueue. Threads can use a socket's wait queue to wait for events occurring on that socket; more than one thread can wait for a socket at any given time.

Threads waiting on a listener socket are queued on the socket's waitqueue. These threads have put themselves to sleep, yielding to other threads. When the network layer detects a new connection request for the listener, it will awaken any threads on the listener socket's wait queue via a socket call-back.

6.1 Socket callbacks

A *socket callback* is a virtual function that the kernel network stack invokes to signal a socket state change to higher-level modules. TCP sockets generally use the default socket callbacks that wake sleeping processes and generate appropriate user-level POSIX signals.

- `sock_def_wakeup()` – signals a generic socket state change.
- `sock_def_error_report()` – signals that some in-band or out-of-band error occurred on the socket.
- `sock_def_readable()` – signals that data is available to be read from the socket.
- `sock_def_write_space()` – signals that buffer space is available for more write operations.
- `sock_def_destruct()` – invoked to release any protocol-specific storage before a `sock` data structure is freed.

Threads normally sleep on waitqueues while waiting for something interesting to happen. When work on an existing connection arrives, the kernel removes the threads from the associated socket's wait queue and the *default readable* callback (`sock_def_readable` for regular TCP sockets) is invoked.

For sockets that TUX uses, the socket callbacks are replaced with new functions implemented by TUX. The old function pointers are saved in case the socket is redirected, in which case the old function pointers are restored before the redirection completes.

`idle_event()` is invoked from these callbacks when a socket changes state. It adds incoming requests to a thread's input queue. In this way, the network layer drives the web server's work by waking up its threads whenever something needs to be done. Molnar inserted `idle_event()` into both the default socket callbacks and into the TUX socket callbacks. We're not certain why it is needed in both places.

TUX bypasses normal `accept()` processing. Work is moved directly from the listener socket's queue to the thread's input request queue. Comments in the code suggest there is some inefficiency in the current Linux `accept()` implementation that is avoided by TUX's new connection accept logic.

6.2 Redirecting connections to the backing server

The main event loop for fast threads visits the redirection queue during each pass. Thus each fast thread is responsible for handling its own redirection requests.

Tux places the socket to be handed off directly onto the backing server's accept queue. It uses the backing server's port number to look up its listen queue. Currently, the backing server must run on the same host with TUX.

TUX replaces a socket's callback functions with functions that are specific to the TUX HTTP layer. If a socket is redirected, these functions are replaced again with the normal TCP layer callback functions.

6.3 Request scheduling

Each thread manages several queues of requests. Request structures are more than half a page, so when a request structure is released, it is saved on a per-thread free list and re-used.

Each thread has seven scheduling queues, anchored in the thread's `threadinfo` object described in the previous section.

- *All requests* – Whenever a new request structure is allocated, it goes on this list. In-use request structures are always on this queue; they may or may not be on one of the other work queues.
- *Free requests* – When a request is finished, the request structure can be re-used. Reusable structures are placed on this list; in-use request structures never appear in this list.
- *Input requests* – Requests waiting for incoming work from a client are placed on this list. When a connection is accepted, `accept_requests()` creates a new request and adds it to the input queue. When a socket state change occurs on a HTTP connection, `idle_event()` adds the request associated with the socket to this queue.
- *User space requests* – User-level work is queued here by the `http()` system call.
- *Output requests* – When a thread requests that a response be returned to a client, that request is queued on this list.
- *Redirection requests* – When TUX decides it cannot handle a request it sets the request's `redirect_secondary` field, causing the event scheduler to queue the request on this list.
- *Finishing* – This queue does not appear to be used.

When a user-level TUX thread calls into the kernel specifying the HTTP_ACTION_EVENTLOOP action, the thread enters its main event loop. The event loop tries to accept new connections by calling `accept_requests()`, checks for work on the thread's five active request queues, then checks for pending signals. The loop finishes by checking if other parts of the kernel want to preempt the thread. If the thread isn't preempted, it will loop to pick up more work; otherwise it invokes the scheduler so other threads can run. If there is no more work, it will sleep.

In several areas, TUX tries to fulfill a client request as soon as possible. During accept processing in `accept_requests()`, TUX attempts to respond immediately to client requests which have already arrived in their entirety. In addition, when parsing headers, TUX can immediately redirect or queue an output request in response to a client request.

In summary, all HTTP sockets are associated with a thread when they are created via TUX's special connection accept logic. Incoming work on a socket is queued onto a thread's input request queue by special socket callbacks invoked from the kernel's network layer. When the thread passes through its main event loop, it parses the input request and requeues the request according to the work that needs to be done.

7. VM Modifications

Some modifications to the kernel's memory management functions were necessary to increase the amount of real and virtual memory available to TUX. Web serving can be a memory-intensive operation.

7.1 Size of kernel address space increased

Normally, the Linux kernel shares the top 1G of process address space with all processes on the system. On systems with 4G of physical addressability, this provides 3G address spaces for each process. The division of kernel space and address space is controlled by a compile-time constant. The TUX patch lowers this constant to increase the kernel's address space size to 3G. This allows more cached data to be addressable inside the kernel.

7.2 System cache reaping

Molnar added several changes to the directory entry cache, the quota cache, and the inode cache logic to make them more selective about where and when to recycle cache items. The authors do not believe these changes are related to the operation of TUX.

An additional interface was added to the page cache to allow TUX to start flushing pages for files that are very large. This is used to maintain a cap on the size of cached files, and also for flushing log data.

TUX wraps the kernel's generic memory allocator in a function called `http_kmalloc()` which is used internally by TUX and trusted dynamic modules. In addition to allocating memory, this function recovers from short-term memory shortages by directly pruning the inode and directory entry caches. This is special behavior copied from logic that runs in the system swap daemon and the system page allocator.

As you may recall, the directory entry cache is used to cache web objects. By pressuring this cache when `http_kmalloc()` discovers a memory shortage, TUX is controlling the web object cache size with direct feedback.

7.3 Atomic directory entry cache lookups

The directory entry cache normally populates its own cache when an entry lookup request fails. TUX adds an option to the directory entry lookup interface to request a lookup operation that examines the cache and returns if no matching entry is found. This way TUX won't block if it uses the directory entry cache, and can populate the missing cache item itself using its own asynchronous I/O mechanism.

User-level applications also get access to this new style of lookup. A new option for `open()`, `O_ATOMICLOOKUP`, specifies that the lookup operation should not block.

7.4 Other miscellaneous changes

In this subsection we list several minor changes included with the TUX patch. These are changes that improve system scalability and performance in general.

The interface that manages inodes with zero reference counts, known as `iput()`, is now split into two interfaces. One places a reusable inode at front of the inode cache's reuse queue, the other at the tail of the queue. This allows parts of the kernel to specify that an inode should be recycled immediately, for example, when the file it represented is deleted, to save room in the inode cache for other inodes that represent files that may be reused in the near future.

The kernel's mechanism to put information into the system log and onto the system console is `printk()`. The TUX patch increases the size of the `printk` log buffer by eight times. This is intended as a debugging aid, but will also help system scalability as faster systems generate log data at a higher rate.

In Linux's page cache and disk quota cache, reclamation goals control how many pages and quota structures are reclaimed when system memory is exhausted. The TUX patch raises these goals so that more pages and quota structures are reclaimed during each reclamation pass through these caches.

Linux divides physical memory into several zones, such as DMA-only pages, or pages that are above 4G. Each zone is balanced on a priority basis when system memory is exhausted. The DMA zone is smallest, yet is important for I/O requests, so it is balanced carefully. The TUX patch eliminates the zone balancing priorities for memory above 4G. This probably has few effects on the zone-balancing algorithm.

Active disk requests in the Linux kernel are queued on an asynchronous request queue. Periodically, Linux will run down this queue to force the disk devices to begin working on outstanding requests. The TUX patch makes the Linux scheduler more aggressive about flushing the active disk request queue. Threads invoke the `schedule()` function when putting themselves to sleep. This change causes most calls to `schedule()` to flush the queue of asynchronous disk requests. The old scheduler never touched the active disk request queue.

To improve the efficiency of sending packets out onto the network, the network layer must wait appropriately for the outgoing message to be complete. The `MSG_NO_PUSH` option, added by the TUX patch, can cause the network layer to hold onto an outgoing packet rather than aggressively push it onto the network. This allows packets to be built in stages, and it allows separate packets to be sent together in a single network write operation.

The Linux network layer uses a hash table to manage incoming TCP connections that are in early stages of connection. The TUX patch increases the size of this hash table eight-fold to help reduce the length of the hash chains in this table when a large number of incoming connections arrive concurrently.

The TUX patch also increases the maximum backlog of unprocessed incoming network packets by three orders of magnitude. Up to 300,000 packets can wait for processing before the protocol-independent networking logic begins dropping packets.

TUX raises the default listen backlog for large machines from 1024 to 4096.

In summary, the TUX patch modifies basic kernel VM functions to facilitate its use (reuse) of the directory entry cache. It also increases the size of the kernel's portion of the virtual address space.

8. Conclusions and Future Efforts

The TUX web server gains considerable performance and scalability improvements by moving oft-used web server functionality closer to the kernel's networking stack. By driving web service directly with incoming network events, and by keeping cached data in the network layer, TUX can respond to web clients faster than more traditional web servers.

TUX supports some advanced features, such as server-side includes, a dynamic content API in the kernel, and SMP scalability. However, TUX is still missing functionality. It doesn't work well with proxy caches, lacks support for virtual hosting, and is missing some needed security checking in the user-level interface.

Some of TUX's architectural ideas are significant and can be directly useful in other operating systems. However, the specific design of the Linux networking stack (*i.e.* socket callbacks) allows an event-driven HTTP handler in a way that may be less portable than other design concepts.

TUX is a large and complex set of modifications to the Linux kernel. In order to gauge the effectiveness of some of its design features, the TUX patch should be broken apart into smaller parts that can be studied more scientifically. For instance, careful analysis of network buffer manipulations might demonstrate areas for improvement, or prove the method is valuable for other types of network servers.

Rigorous security analysis of the `http()` system call and the new HTTPAPI is necessary before determining that such an architecture is appropriate for high-performance and secure installations.

Finally, a lock contention study might show that TUX performs well with fewer locks. Linux spin locks are designed for low instruction count in the case where the lock isn't already held. However, extra locks result in cache line sharing in SMP configurations, which slows memory traffic for all work on the system.

8.1 Acknowledgements

Special thanks go to Peter Honeyman and Andy Adamson at U-M, and to Will Morris at iPlanet. Special thanks to Niels Provos for ideas and encouragement.

9. References

1. W. Richard Stevens, *Advanced Programming in the UNIX Environment*, Addison-Wesley Publishing Co., Copyright 1992. ISBN 0-201-56317-7.
2. Ingo Molnar, "TUX web server 1.0," `people.redhat.com/mingo/tux-2.4.0-test8-C4` .
3. "Second Quarter 2000 SPECweb99 Results," `www.spec.org/osg/web99/results/res2000q2/`
4. Ingo Molnar, "Answers from Planet TUX: Ingo Molnar Responds," Slashdot, July 20,2000, `slashdot.org/interviews/00/07/20/1440204.shtml` .
5. Apache Server, The Apache Software Foundation. `www.apache.org`
6. Jun-ichiro itojun Hagino, "Mbuf Issues in 4.4BSD IPv6 Support—Experiences from KAME IPv6/IPsec Implementation," *Proceedings of the FREENIX track, 2000 USENIX Technical Conference*, June 2000.
7. Craig Metz, "Porting Kernel Code to Four BSDs and Linux," *Proceedings of the FREENIX track, 1999 USENIX Technical Conference*, June 1999.
8. Vivek S. Pai, Peter Druschel, Willy Zwaenepoel, "Flash: An Efficient and Portable Web Server," *Proceedings of the 1999 USENIX Technical Conference*, June 1999.
9. *Internet Information Services 5.0 Programmer's Guide: ISAPI Reference*, MSDN Online Library, Copyright 2000, Microsoft Corporation.
10. J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, T. Berners-Lee, *Hypertext Transfer Protocol -- HTTP/1.1*, RFC 2616, June 1999.
11. Arjan van de Ven, "khttpd: Linux HTTP Accelerator," `http://www.fenrus.demon.nl/`
12. V. S. Pai, P. Druschel, W. Zwaenepoel, "IO-Lite: A unified I/O buffering and caching system," *ACM Transactions on Computer Systems*, Vol. 18, No. 1, pp.37-66, February 2000.